Prioritization of Exploit Codes on GitHub for Better Vulnerability Triage

Kentaro Kita[1], Yuta Gempei[1], Tomoaki Mimoto[1], Takamasa Isohara[1], Shinsaku Kiyomoto[1] and Toshiaki Tanaka[2]

[1]*KDDI Research, Inc., 2-1-15 Ohara, Fujimino-shi, Saitama, Japan*

[2]*University of Hyogo, 7-1-28, Minatojima-minamimachi, Chuo-ku, Kobe, Hyogo, Japan*

{*ke-kita, yu-genpei, to-mimoto, ta-isohara, sh-kiyomoto*}*@kddi.com, toshi@gsis.u-hyogo.ac.jp*

Keywords: Threat Intelligence, Vulnerability, Exploit Code, GitHub.

Abstract: Analyzing exploit codes is essential for assessing the severity of vulnerabilities and developing effective defense measures against future exploits. Whereas ExploitDB and Metasploit are two major sources of exploit codes, GitHub has been rapidly growing into a promising platform for sharing exploit codes. However, prioritizing GitHub exploit codes to be analyzed in depth is challenging, owing to its large collection of codes and the absence of mechanisms for guaranteeing the validity of codes published by users. To address this problem, this paper proposes a scheme to prioritize GitHub exploit codes based on their source codes and repository metadata. First, we show that GitHub often contains different but semantically similar exploit codes targeting the same vulnerability, and such duplicated codes can be efficiently removed with code clone detection techniques. Second, we leverage a feature of GitHub that it plays the role of a social networking platform. By mining a graph that represents relationships among GitHub users, our scheme prioritizes exploit codes by taking both the reputation from users and security community's attention to targeted vulnerabilities into consideration.

1 INTRODUCTION

In the rapidly evolving landscape of cybersecurity, analyzing exploit codes (or proof-of-concept codes), which are program codes to exploit specific vulnerabilities, plays a crucial role for vulnerability triage. In-depth analysis of exploit codes allows Security Operation Center (SOC) analysts to prioritize high-risk vulnerabilities for their organizations, thereby allowing them to take defensive measures for mitigating future exploits. In addition, public availability of sophisticated exploit codes indicates that potential attackers even without professional-level knowledge are likely to be able to exploit the vulnerabilities successfully. For these reasons, Common Vulnerability Scoring System (CVSS) (Forum of Incident Response and Security Teams (FIRST), 2023) and Exploit Prediction Scoring System (EPSS) (Jacobs et al., 2023) use information regarding the existence and maturity of exploit codes for vulnerability assessment.

ExploitDB and Metasploit are public sources of exploit codes widely used for a long time by security analysts, researchers, and penetration testers. ExploitDB is an archive of exploit codes maintained by OffSec and its content relies on contributions from users. In contrast, Metasploit is an open-source penetration testing framework developed by Rapid7. It provides a large set of pre-configured exploit codes as its modules, allowing users to customize and simulate attacks against known vulnerabilities.

In addition to ExploitDB and Metasploit, there have been many exploit codes shared on GitHub in recent years. However, there are no existing studies that thoroughly compare GitHub with ExploitDB and Metasploit in terms of their exploit codes. This leads us to study the following research question: Are the exploit codes published on GitHub worth collecting and analyzing for vulnerability triage compared to ExploitDB and Metasploit? To answer this question, we compare exploit codes published on GitHub, ExploitDB, and Metasploit using the following four metrics: the number of exploit codes, coverage of vulnerabilities, coverage of exploits in the wild, and timeliness of exploit codes publication. Our results reveal that GitHub is indispensable for comprehensive assessment of vulnerabilities.

To realize efficient vulnerability triage, it is desired to automatically extract exploit codes that worth in-depth analysis by humans or computers among numerous codes on GitHub; however, this is challeng-

Kita, K., Gempei, Y., Mimoto, T., Isohara, T., Kiyomoto, S. and Tanaka, T.
Prioritization of Exploit Codes on GitHub for Better Vulnerability Triage.
DOI: 10.5220/0013100800003899

In *Proceedings of the 11th International Conference on Information Systems Security and Privacy (ICISSP 2025) - Volume 1*, pages 27-38
ISBN: 978-989-758-735-1; ISSN: 2184-4356

ing because GitHub is not dedicated for sharing exploit codes unlike the other two sources. First, for a single vulnerability, users often publish different but semantically similar exploit codes. Manually inspecting such duplicated exploit codes is very time-consuming. Second, GitHub has no mechanisms to guarantee the validity of exploit codes. In contrast, ExploitDB has a mechanism for its maintainers to explicitly mark exploit codes whose validity have been verified as "verified". In this paper, we present our first attempt to design a scheme to prioritize exploit codes based on their source codes and repository metadata.

First, we show that duplicated exploit codes can be identified with reasonable accuracy by combining existing code clone detection tools, SourcererCC (Sajnani et al., 2016) and ssdeep (Kornblum, 2006). Our evaluation reveals that at least one clone codes have been published for 58.5% of vulnerabilities with multiple exploit codes and our scheme can effectively narrow down candidates of exploit codes to be further analyzed.

Second, we design a scheme to extract trustworthy exploit codes by leveraging a unique feature of GitHub that it plays the role of a social networking platform. The key insights behind our design are twofold: (1) We can construct a graph that represents trust and interests among GitHub users by using the metadata of users and their repositories, such as followers of users and stars assigned to repositories; and (2) We can identify a set of GitHub users whose exploit codes are valid with high confidence by using external information sources including the NVD CVE database, ExploitDB, and Metasploit. Specifically, we regard GitHub users whose exploit codes are referenced by any of the NVD CVE records and those who have published valid exploit codes on ExploitDB or Metasploit as roots of trust in the graph. We then determine the trustworthiness of exploit codes by using a graph mining technique called TrustRank (Gyöngyi et al., 2004).

The contributions of this paper are as follows:

- By systematically collecting and investigating exploit codes on GitHub, ExploitDB, and Metasploit, we show that the significance of monitoring exploit codes published on GitHub has been increasing in recent years.
- We show that GitHub contains a lot of different but semantically similar exploit codes and such duplications can be effectively identified and removed by analyzing their codes.
- We design and implement a graph mining-based scheme for prioritizing trustworthy exploit codes based on social relationships among GitHub users. A key feature of our scheme is to leverage the NVD CVE records and exploit codes on ExploitDB and Metasploit to improve the reliability of the results. To the best of our knowledge, this is the first study to design a scheme for prioritizing GitHub exploit codes.

The rest of the paper is organized as follows: Section 2 describes our methodology to collect exploit codes from GitHub, ExploitDB, and Metasploit. Section 3 quantitatively compares these exploit code sources. Section 4 describes our scheme to prioritize GitHub exploit codes. Section 5 summarizes related works. Finally, Section 6 concludes this paper.

2 DATA COLLECTION

In the rest of the paper, we only focus on vulnerabilities that assigned CVE IDs by MITRE. CVE records marked as "REJECTED" or "RESERVED" are excluded. In this section, we describe the features of three major sources of exploit codes: GitHub, ExploitDB, and Metasploit, and our methodology for collecting exploit codes.

2.1 GitHub

GitHub is a web-based software development platform. Each project is managed as a repository that contains various files including source codes, binary codes, and text files. An important feature of GitHub is that it also acts as a social networking platform. Users can follow other users to stay updated on their activities and projects. They can star repositories to bookmark them for future reference. Watching a repository allows users to receive notifications about its updates. Forking creates a copy of another user's repository, allowing users to apply experimental changes without affecting the original projects. These features facilitate a highly interactive and connected community, encouraging collaboration and code sharing.

It is more difficult to correctly extract exploit codes correlated with a specific CVE ID on GitHub than ExploitDB and Metasploit because GitHub hosts a much larger number and variety of codes. Simply retrieving GitHub repositories using a CVE ID as a keyword can return repositories that contain scripts to exploring Indicator of Compromise (IoC) on devices to check if they have been attacked. To address this problem, we combined several heuristics under the assumption that repository titles, descriptions, tags, readme files are reasonably informative.

First, we used the GitHub Search API to retrieve repositories that mention a specific CVE ID in their titles and descriptions. As a result, we obtained 12638 repositories. Note that forked repositories are omitted from the search results.

Second, we checked if the repositories are correlated with the same CVE IDs as those specified in the search queries. For example, if we specify CVE-2023-1111 in a search query, the response contains not only the exploit codes correlated with `CVE-2023-1111` but also those correlated with `CVE-2023-11110`, `CVE-2023-11111`, and so on. To address this problem, we extracted CVE IDs from the metadata of the repositories, such as title, description, tags, and readme file using regular expressions, and then excluded entries that did not contain the specified CVE IDs.

Third, we excluded repositories whose metadata contains more than three different CVE IDs because such repositories often contain multi-purpose vulnerability detection/exploitation tools (e.g., NSE scripts for Nmap) or text files that merely enumerate various vulnerabilities and relevant URL links. We did not exclude repositories containing two CVE IDs because exploit codes often aim to increase impacts of attacks by exploiting multiple vulnerabilities simultaneously. Recent examples include the exploitation of `CVE-2023-46805` (an improper authentication vulnerability of Ivanti products) and `CVE-2024-21887` (a command injection vulnerability of the same products) that allows unauthenticated remote attackers to execute arbitrary commands.

Fourth, we excluded repositories whose metadata did not contain keywords indicating that they contain exploit codes, such as "exploit", "proof-of-concept", "checker", and "scanner". We also excluded repositories that contain keywords indicating that they do not contain exploit codes, such as "indicator of compromise", since such repositories contain IoC scanners for potentially compromised hosts.

Finally, we excluded repositories whose language tags automatically assigned by GitHub are not contained in a list of popular programming languages for developing exploit codes. The list consists of C, C#, C++, Go, Java, JavaScript, Nim, PHP, Perl, PowerShell, Python (including Jupyter Notebook), Ruby, Rust, shell script. This rule aims to exclude empty repositories and repositories that just contain readme or text files detailing specific vulnerabilities.

After designing this heuristics-based scheme, we evaluate its performance with our manually labeled dataset of 144 GitHub repositories for randomly chosen 17 vulnerabilities published in 2023. Among them, 13 repositories do not contain exploit codes. For this dataset, the precision, recall, and F1 score were 0.972, 0.786, and 0.869, respectively. Although the recall value is relatively low, we argue that this is a good balance to avoid overestimating the number of exploit codes on GitHub in Section 3. In total, we collected 4537 exploit codes from GitHub.

2.2 ExploitDB

ExploitDB is a public archive of exploit codes and shell codes maintained by OffSec. Each exploit code is submitted by users with its metadata including the title, author, affected software, and the corresponding CVE IDs (if applicable). One of the features that make ExploitDB a useful source of exploit codes is the fact that it includes a mechanism to guarantee the validity of their exploit codes. Newly submitted exploit codes are first marked as "unverified" and then undergoes verification process by maintainers. If exploit codes pass the verification process, they are explicitly marked as "verified" to allow users to easily choose functional exploit codes.

To extract exploit codes on ExploitDB that target the vulnerability with a specific CVE ID, we used SearchSploit, an official command-line tool to retrieve exploit codes on ExploitDB using flexible keyword-based search. For example, to retrieve exploit codes targeting the vulnerability identified with `CVE-2023-1111`, we can use the following command: "searchsploit –cve 2023-1111". However, this query alone is not sufficient to extract exploit codes exactly matching the specified CVE ID. We therefore excluded a repository if its metadata did not contain the same CVE ID as specified in a search query in the same way as described in Section 2.1. In total, we collected 15119 exploit codes from ExploitDB.

2.3 Metasploit

Metasploit is an open-source penetration testing framework widely used for security assessments. It provides a vast range utilities including exploit codes, shell codes, vulnerability scanner, and post-exploitation codes as its modules. These modules are pre-configured scripts designed to breach specific weaknesses, allowing users to easily simulate attacks. Each module can be paired with different payloads to achieve various post-exploitation objectives.

To obtain a list of exploit codes (exploit modules) of Metasploit, we installed Metasploit v6.4.12 on a local machine and extracted all the metadata regarding Metasploit modules including exploits, auxiliary, post, payloads, encoders, and evasion. After that, we correlate each exploit code with a specific CVE ID if

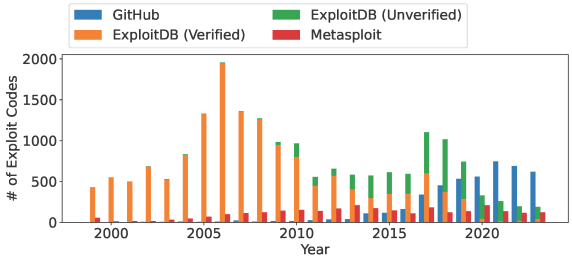


Figure 1: The number of exploit codes published on GitHub, ExploitDB, and Metasploit for vulnerabilities found in each year.

the module name, description, or reference section of its metadata contains the CVE ID. In total, we collected 2425 exploit codes from Metasploit.

3 COMPARISON OF SOURCES OF EXPLOIT CODE

In this section, we compare GitHub, ExploitDB, and Metasploit in terms of the following metrics that indicate whether each of the sources can provide information on a wide range of vulnerabilities in a timely manner: (1) the number of exploit codes, (2) coverage of CVE IDs, (3) coverage of exploits in the wild, and (4) timeliness of exploit code publication.

3.1 Number of Exploit Codes

Fig. 1 presents the numbers of exploit codes published on the three sources for vulnerabilities found in each year. Since ExploitDB contains verified codes and unverified codes, they are shown separately.

First, the number of exploit codes published on ExploitDB is largest among the three sources for vulnerabilities found between 1999 and 2019. However, the number has been declining recent years. The ratio of verified exploit codes has also been declining; $> 50\%$ between 1999 and 2017 and $< 25\%$ between 2020 and 2023. This could be because the verification of newly submitted exploit codes is still in queue of ExploitDB maintainers. We therefore conclude that ExploitDB is particularly useful for those who are interested in exploit codes published a few years ago.

Second, GitHub has few exploit codes until 2014; however, it is the largest source of exploit codes between 2020 and 2023. The number of exploit codes has been increasing, and more than 600 exploit codes have been published in recent years. Note that multiple GitHub users can publish exploit codes for the same vulnerability, and we count them individually. Thus, this result does not mean that exploit codes on GitHub cover more vulnerabilities.

Different from the other two sources, the number of exploit codes on Metasploit is stable regardless of the years (min: 14 codes, max: 210 codes).

3.2 Coverage of CVE IDs

Fig. 2 presents Venn diagrams that represent the coverage of CVE IDs, defined as the number of vulnerabilities for which each source has exploit codes. The higher the number, the greater the availability of codes for a wide range of vulnerabilities.

Overall, the coverage of ExploitDB is almost six times larger than the other two sources. This result indicates that ExploitDB is a mature platform for sharing exploit codes and is the best source for security researchers who want to conduct more comprehensive vulnerability monitoring.

In terms of the changes in the coverage over time, we can see that the coverage of GitHub has increased rapidly in recent years. This is consistent with the fact that the number of exploit codes published on GitHub has been increased, as described in Section 3.1. In particular, GitHub is the best source in terms of the coverage of vulnerabilities found in 2023. This result shows that the importance of monitoring exploit codes on GitHub has been increasing in recent years.

Another notable result is that the intersections of vulnerabilities covered by each pair of the sources are small. Specifically, only 12.5% of the vulnerabilities have exploit codes published on multiple sources. In other words, there are many vulnerabilities that are only covered by a single source.

3.3 Coverage of Exploits in the Wild

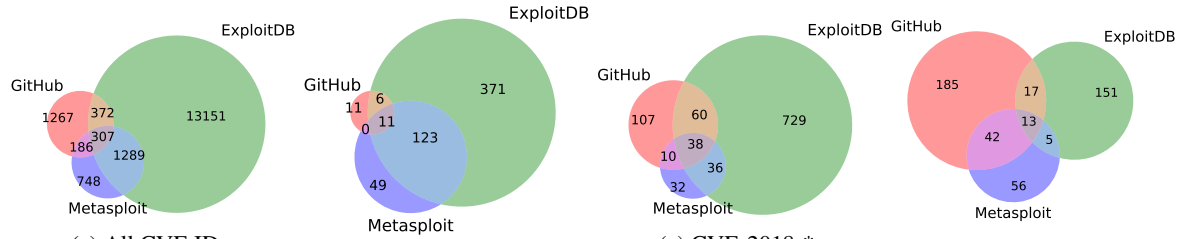
In addition to the coverage of CVE IDs, the coverage of vulnerabilities that has been exploited in the wild is an important measure to understand how much information on the realistic cyberattacks each of the sources provides.

To the best of our knowledge, there is no comprehensive dataset of vulnerabilities that have been exploited in the wild. Therefore, we collected the following three pieces of information to create a list of vulnerabilities exploited in the wild:

1. Known Exploited Vulnerabilities (KEV) catalog
2. Microsoft Security Response Center (MSRC) exploitability indexes
3. Symantec attack signatures

First, the KEV catalog[1] is a list of vulnerabilities with active exploits reported to Cybersecurity and In-

[1]https://www.cisa.gov/known-exploited-vulnerabilities-catalog



(a) All CVE IDs. (b) CVE-2013-*. (c) CVE-2018-*. (d) CVE-2023-*.

Figure 2: Coverage of CVE IDs.

Table 1: Coverage of exploits in the wild.

Code Source	# of vulnerabilities	Ratio
GitHub	453	32.9%
ExploitDB	432	31.3%
Metasploit	472	34.3%
Overall	756	54.9%

frastructure Security Agency (CISA). We regard vulnerabilities that have been added to the KEV catalog at least once as exploited in the wild. One of criteria for a vulnerability to be added to the KEV catalog is that there must be a clear remediation guidance for organizations using the affected products. Thus, the KEV catalog might not cover all the vulnerabilities exploited in the wild.

Second, MSRC investigates vulnerabilities of Microsoft products and publishes the results as the Security Update Guide[2]. In the results, exploitability of each of the vulnerabilities is represented as an exploitability index, which takes one of the following four values: "Exploitation detected", "Exploitation more likely", "Exploitation less likely", and "Exploitation unlikely". We extracted vulnerabilities marked as "Exploitation detected".

Third, Symantec is publishing a list of signatures currently monitored by their security products as Symantec Attack Signatures[3]. Allodi et al. have shown that the CVE IDs mentioned in the signatures are the best available indicator for estimating the existence of exploit codes correlated to a specific CVE ID (Allodi and Massacci, 2012). Other studies have also used the list of CVE IDs as their ground truth for creating datasets of vulnerabilities exploited in the wild (Sabottke et al., 2015; Suciu et al., 2022).

By removing duplicates from the three sources, we extracted 1378 vulnerabilities exploited in the wild. Table 1 summarizes the number and ratio of vulnerabilities exploited in the wild for which exploit codes are available. The source with the largest ratio is Metasploit, which covers 34.3% of the vulnerabilities, but GitHub and ExploitDB are comparable to it. This result shows that Metasploit is the best source for analyzing particularly high-risk vulnerabilities.

Another noteworthy result is that the intersections of vulnerabilities covered by these sources are small. Only about 32% of vulnerabilities exploited in the wild can be covered if one of the sources is monitored. In contrast, more than half of the vulnerabilities can be covered if we combine all of them.

3.4 Timeliness

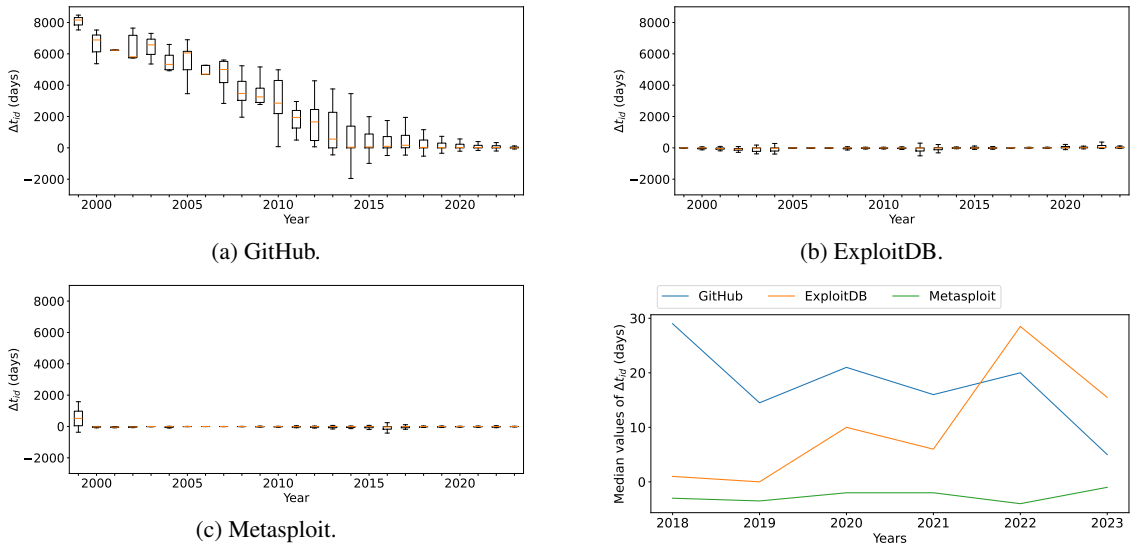
We define timeliness as the dates it takes until the first exploit code for a specific vulnerability is published after the vulnerability is made public, assuming that the vulnerabilities were widely known to the public at the publication date of the CVE record. Specifically, for every vulnerability with CVE ID *id*, t_{id} is defined as the difference between the date the first exploit code correlated with *id* is published and the date the CVE record for *id* is published. t_{id} becomes negative if an exploit code is published before the corresponding CVE record is published.

It is difficult to reliably determine the publication date of a CVE record because the publication dates for a specific CVE ID sometimes different significantly between MITRE and NVD. For example, `CVE-2003-0590` is reserved at July 17th, 2003 by MITRE but the CVE record was not published until October 17th, 2016. In contrast, NVD published the corresponding CVE record much earlier: on August 18th, 2003. As another example, the NVD CVE record for `CVE-1999-1557` was published on May 2nd, 2005 but the MITRE CVE record was published on August 31st, 2001. These large differences reduce the reliability of t_{id} values. To obtain conservative results, we define the earlier one among the publication dates of CVE records by MITRE and NVD as the date when the CVE record was published.

Fig. 3a, Fig. 3b, and Fig. 3c present box plots of t_{id} values. The orange lines in the box plots represent the median values. Following the definition of box plots, t_{id} that are far from the interquartile ranges

[2]https://msrc.microsoft.com/update-guide/vulnerability

[3]https://www.broadcom.com/support/security-center/attacksignatures



(a) GitHub.

(b) ExploitDB.

(c) Metasploit.

(d) Median values of t_{id}.

Figure 3: Timeliness.

are regarded as outliers and not plotted in the figures. Fig. 3d presents the distributions of the median values of the three sources between 2018 and 2023.

As shown in Fig. 3b and Fig. 3c, the median values of ExploitDB and Metasploit are in the range of $[-10, 20]$ for almost all the years. This implies that, for many vulnerabilities, analysis of exploit codes on ExploitDB and Metasploit provides valuable information shortly after (or even before) the vulnerabilities are publicly disclosed. In contrast, as shown in Fig. 3a, the median values of GitHub were tens or hundreds of times larger between 1999 and 2017. This is because most of the GitHub exploit codes were published after 2018. However, as shown in Fig. 3d, the timeliness of GitHub between 2018 and 2023 have been comparable to the others. For example, in 2023, the median values of GitHub, ExploitDB, and Metasploit are 5.0, 15.5, -1.0, respectively. This could be because GitHub have become widely known as a promising platform for sharing exploit codes.

Finally, we manually investigated some of the outlier values of t_{id} and found that their causes include the fact that it might take a few years from the discovery of a vulnerability to the publication of the corresponding CVE record. For example, the vulnerability identified with `CVE-2018-7935` (a DoS vulnerability of Huawei mobile routers) was discovered in 2018 and an exploit code was published on GitHub by the discoverer[4]; however, the CVE record and vendor advisory were published in 2023[5]. Similarly, `CVE-2021-31796` was discovered and an exploit code was published on GitHub in 2017[6]; however, the publication of the CVE record was in 2021.

3.5 Summary of Results

The observations obtained in this section can be summarized as follows:

- Until a few years ago, the significance of monitoring exploit codes on GitHub was low; however, it has begun to attract the attention from the security community as a promising platform for sharing exploit codes and has been rapidly growing into a source of exploit codes comparable to ExploitDB and Metasploit in term of the number of exploit codes, coverage of CVE IDs, coverage of exploits in the wild, and timeliness.
- The intersections of vulnerabilities covered by GitHub, ExploitDB, and Metasploit are small. This implies that there is an urgent need to collect and monitor exploit codes published on all the sources rather than focusing on a single source for more comprehensive monitoring of exploit codes. However, such augmentation of coverage substantially increases the burden of exploit code analysis process by SOC analysts of organizations. Therefore, it is desired to design and implement a scheme to (semi-)automatically prioritize exploit codes that worth in-depth analysis from the GitHub's large collection of codes.

[4]https://github.com/lawrenceamer/CVE-2018-7935

[5]https://www.huawei.com/en/psirt/security-notices/huawei-sn-20230210-01-dos-en

[6]https://github.com/unmanarc/CACredDecoder

4 PRIORITIZING GitHub EXPLOIT CODES

In Section 3, we have revealed that we need to collect and analyze not only exploit codes on ExploitDB and Metasploit but also those on GitHub for more comprehensive monitoring of exploit codes and better vulnerability triage. However, extracting GitHub exploit codes that worth in-depth analysis is more difficult task than one might consider due to the following two problems:

1. The number of exploit codes has been growing and GitHub users can publish multiple exploit codes that targets the same vulnerability. In our GitHub dataset, 30.3% of vulnerabilities have more than two exploit codes. For example, we found 25 exploit codes for `CVE-2022-46169` (a command injection vulnerability of Cacti, which is listed in the KEV catalog). Manually inspecting such duplicated exploit codes one by one is very time-consuming.
2. Unlike the other two sources, GitHub has no built-in mechanisms to guarantee the validity of each exploit code. In contrast, ExploitDB has a mechanism for the maintainers to explicitly mark exploit codes that have been verified to be valid, and Metasploit contains only functional exploit codes as its modules.

In this section, we present our first attempt to design a scheme to prioritize exploit codes based on their source codes and metadata for triaging high-risk vulnerabilities. First, we show that GitHub repositories often contain very similar exploit codes for the same vulnerability and such duplicated codes can be removed with reasonable performance by using code clone detection techniques. Second, we design a graph-based scheme to extract trustworthy exploit codes by leveraging several mechanisms of GitHub to represent interests and trust among users, such as stars and forks.

4.1 Removing Similar Exploit Codes with Code Clone Detection

For detecting semantically similar exploit codes with code clone detection tools, we focus on Python exploit codes because Python is the most widely used language for exploit code. Among 4537 GitHub repositories, 56.0% are Python, 11.8% are C, 7.0% are shell script, 4.8% are Java, and 4.7% are C++.

Before exploring code clones among the entire Python exploit codes, we evaluate the performance of existing clone detection techniques with our manually labeled dataset of 144 Python exploit codes for 17 vulnerabilities published in 2023. For clone detection, we focused on Type-1, Type-2, and Type-3 code clones (Bellon et al., 2007); in brief, two codes are regarded as a clone pair if they are identical except for some modifications that do not affect their behavior (e.g., whitespaces, comments, variable/function identifiers) or if they are copied codes with further modifications such as changed, added, or removed statements. In our dataset, 13 of the 17 vulnerabilities have at least one pair of cloned exploit codes.

A unique feature of exploit codes is that they often contain many statements to display code metadata or debug information to improve their usability. A typical example is the `print` function used to display banners that contain author names and version information. However, these statements have no practical effects on their functionality. In addition, different authors often use very different output strings, resulting in negative effects on clone detection. To mitigate this problem, we removed function calls to output strings to stdout/stderr, including `print` and `logging.debug` in advance.

Table 2 summarizes our main performance evaluation results, where θ represents clone threshold, i.e., two exploit codes are regarded as a clone pair if their similarity score is larger than θ. The first candidate is a BERT-based Python clone code detection model (sangHa0411, 2022; lazyhope, 2023) trained with a public dataset (PoolC, 2021). BERT-based code analysis models have been actively studied in recent years and achieve state-of-the-art performance for several tasks (Feng et al., 2020). However, the performance of the BERT-based model is not satisfactory; it outputs high similarity scores (> 0.99) for almost all the pairs of codes, and as a result, the precision scores become very small. One of the reasons is a feature of BERT-based models that they use small chunks of codes (typically functions or methods) for model training because BERT imposes restrictions on the number of input tokens. Consequently, they are not suitable for applying to exploit codes.

Next, we evaluate SourcererCC (Sajnani et al., 2016), which is a clone detector based on comparison code tokens optimized for large code bases. For our dataset, similarity threshold $\theta = 0.5$ performs the best; however, the recall score is still small (0.678). We therefore incorporate SourcererCC with ssdeep (Kornblum, 2006), a context triggered piecewise hashing scheme commonly used for detecting similar files. Specifically, we regard two exploit codes as a clone pair if SourcererCC or ssdeep determines they are a clone pair. As shown in Table 2, this

Table 2: Performance comparison of code clone detection schemes.

	Accuracy	Precision	Recall	F1
BERT-based model ($\theta = 0.995$)	0.483	0.131	0.745	0.223
BERT-based model ($\theta = 0.999$)	0.684	0.191	0.673	0.298
SourcererCC ($\theta = 0.6$)	0.948	**0.882**	0.545	0.674
SourcererCC ($\theta = 0.5$)	0.951	0.804	0.673	0.733
SourcererCC ($\theta = 0.4$)	0.911	0.536	**0.818**	0.648
SourcererCC ($\theta = 0.5$) + ssdeep	**0.962**	0.804	**0.818**	**0.811**

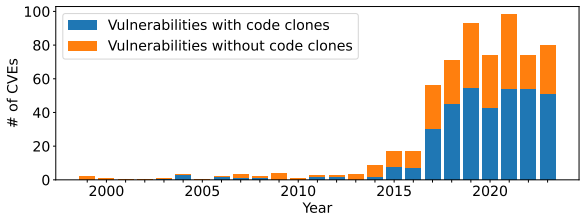


Figure 4: The number of vulnerabilities with at least one code clones among vulnerabilities with multiple exploit codes.

scheme has the best F1 score 0.811.

4.1.1 Results

We explore code clones by applying the above scheme to the entire Python exploit codes. Note that the dataset does not contain forked repositories, as described in Section 2.1. As a result, we found that 58.5% of vulnerabilities with multiple exploit codes have cloned exploit codes. As shown in Fig. 4, code clones are widespread regardless of the years vulnerabilities were found in. The median of the numbers of code clones for each vulnerability is 2 with standard deviation 3.40. `CVE-2023-38646` has 21 clones out of its 23 exploit codes, which is the maximum number of clones for a single vulnerability. The vulnerability with the second most code clones is `CVE-2020-1472` (a.k.a. Zerologon), and 20 out of its 24 exploit codes are determined as code clones.

Next, we discuss the benefits of the code clone detection scheme on reducing the burden of exploit code analysis. We divide all the codes into clone clusters according to the clone detection results. Let us assume that, for analyzing exploit codes of a specific vulnerability, analysts choose only one of exploit codes from every cluster corresponding to the vulnerability. In this case, the number of exploit codes to be analyzed can be reduced by 25.5% compared to the case where all the codes must be analyzed.

These results indicate that code clones are widespread on the exploit code sharing ecosystem on GitHub, and thus detecting them can effectively reduce the number of candidates of exploit codes to be analyzed in depth.

4.2 Graph-Based Exploit Code Prioritization

After identifying clone codes, we further narrow down exploit codes to be analyzed based on their trustworthiness and the attention to the corresponding vulnerabilities. Specifically, we extract repository owners whose exploit codes are valid with high confidence by leveraging TrustRank (Gyöngyi et al., 2004), an algorithm derived from the PageRank algorithm. PageRank was originally designed to derive the influence and importance of web pages; however, it has been applied to the analysis of various platform where users are interconnected, including Twitter (Chien et al., 2014) and GitHub (Hu et al., 2016). A core feature of TrustRank is that it leverages the concept of trust. In brief, it manually chooses a set of trustworthy nodes with trust scores 1.0, and then propagates the scores following the graph structure through multiple iterations until the scores of the nodes converge. Consequently, every node is assigned a trust score in the range of $[0.0, 1.0]$, reflecting its trustworthiness based on the quality and quantity of connections with other nodes.

First, we created a graph that represent relationship between GitHub users. Each node corresponds to a GitHub user and a directed edge from user A to user B is established if any of the following conditions holds: (1) A is following B; (2) A has sent a star for any of the repositories of B; (3) A is watching any of the repositories of B; or (4) A has forked any of the repositories of B. The resulting graph consists of 103210 nodes and 100071 edges. The minimum, median, mean, and maximum degree of the nodes are 1, 1, 5.46, and 4022, respectively.

Next, we created a list of GitHub users who are regarded as roots of trust in the graph, meaning that their exploit codes are assumed to be valid. The first part of the list contains GitHub users whose exploit codes are referenced by any of the NVD CVE records. We collected NVD CVE records for vulnerabilities

published between 1999 and 2023, and then extracted URLs in the reference sections of the records. Among 45678 URLs collected, we found 4378 URLs belong to `github.com`. We further extracted URLs that are identical with the URL of any of the exploit code repositories. As a result, we obtained 190 exploit code repositories referenced by the NVD CVE records, and these repositories belong to 158 users. The second part contains GitHub users who have published valid exploit codes on ExploitDB or Metasploit. We collected the usernames of the authors of Metasploit exploit codes and ExploitDB exploit codes marked as "verified", and associated the authors with GitHub users with exact matching of usernames. To reduce the impact of collisions of usernames among different sources, we excluded users whose usernames are less than four characters. In total, we obtained a list of 280 trustworthy GitHub users. Note that our graph generation process can be fully automated, and thus the rank scores can be easily updated when new GitHub users are added to the graph.

Finally, we derive the rank scores for all the nodes in the graph. The initial trust scores of the trustworthy GitHub users and the other users are set to 1.0 and 0.5, respectively. The scores are then updated iteratively according to the following equation:

$$s^{n+1} = dMs^n + (1-d)s^0,$$

where s^i, M, and d denote the vector of rank scores in the i-th iteration ($i \geq 0$), the transition matrix that represents the graph structure, and the dumping factor, respectively. The damping factor d is set 0.85, which is the default value for the PageRank and TrustRank algorithm. For $M = \{m_{i,j}\}$, $m_{i,j} = 1/(\text{outdegree of node } j)$ if there exists a directed edge from node j to node i and $m_{i,j} = 0$ otherwise. We set the convergence threshold to 10^{-6}, meaning that the iteration process stops if the L2-norm between s^n and s^{n+1} is less than the threshold. To derive accurate rank scores, it is desirable to use a small convergence threshold; however, the amount of time required to derive the scores becomes large. Therefore, the threshold is typically set to 10^{-4} or 10^{-6}. For our graph, the score derivation was completed within 0.304 seconds even when the threshold is set to 10^{-6}.

4.2.1 Results

It takes 46 iterations to derive the rank scores. The distribution of rank scores is long tailed. Specifically, the top 100 GitHub users hold more than half of the sum of the rank scores.

First, we verify if the TrustRank algorithm has sufficient effects on the final ranking of users, compared to the case where users are simply ranked with the number of stars, watchers, forks, or followers. To this end, we use the Pearson correlation coefficient, denoted by $r \in [-1.0, 1.0]$. Table 3 summarizes the correlation coefficient values for all the pairs of the 5 metrics. First, the number of stars, watchers, and forks are positively correlated ($0.760 \leq r \leq 0.956$), while the correlation between the number of followers and the other metrics are small ($0.282 \leq r \leq 0.310$). This is likely because, among the 4 metrics, only the number of followers is the characteristic of users, rather than their repositories. Second, the correlation coefficients between the rank scores and the other metrics are medium ($0.303 \leq r \leq 0.807$). For example, among the top 100 users determined by our TrustRank-based scheme, 52 users are not contained in the top 100 users determined solely by the total number of followers, stars, watchers, forks. From these results, we conclude that our scheme has a sufficient impact on the final prioritization.

Table 4 presents short profiles of the top 5 GitHub users. Three of them are included in the list of trusted owners. The first three owners host exploit codes for famous vulnerabilities of Windows software with codenames including `CVE-2021-1675` and `CVE-2021-34527` (a.k.a. PrintNightmare, remote code execution vulnerabilities of Windows Print Spooler, software responsible for connecting Windows OS with printers and managing printing jobs), `CVE-2022-21999` (a.k.a. SpoolFool, a remote code execution vulnerability of SMBv3), and so on. Similarly, the last two owners publish exploit codes for famous vulnerabilities but not limited to Windows software. One important feature of the exploit codes is that all of them have the top-level user reputation among code targeting the same vulnerability. In other words, our scheme can prioritize exploit code by taking both the reputation from users and the security community's attention to vulnerabilities into consideration.

5 RELATED WORK

Using GitHub for Vulnerability Triage: Existing studies have investigated communication among users on social media platforms including GitHub, Twitter, and Reddit for vulnerability triage. Schiappa et al. have analyzed GitHub events, Tweets, and Reddit posts that explicitly mention CVE IDs using a topic modeling technique (Schiappa et al., 2019). They revealed that these social media platforms, especially Twitter, can be used to identify high-severity vulnerabilities. They also revealed that there is a positive correlation between the number of Twitter posts

Table 3: Correlation coefficient among rank score and other 4 metrics.

	rank score	# of stars	# of watchers	# of forks	# of followers
rank score	1.0	0.807	0.683	0.779	0.303
# of stars	0.807	1.0	0.760	0.956	0.298
# of watchers	0.683	0.760	1.0	0.765	0.310
# of forks	0.779	0.956	0.765	1.0	0.282
# of followers	0.303	0.298	0.310	0.282	1.0

Table 4: The profile of the top 5 GitHub users. We masked their usernames for privacy reasons.

Rank	# of stars	# of watchers	# of forks	# of followers	Content
1	5648	163	1316	1425	8 exploit codes for vulnerabilities of Windows software including PrintNightmare, SpoolFool, SMBGhost, and CurveBall.
2	6236	142	1744	1459	Python exploit codes for PrintNightmare, Zerologon, and vulnerabilities regarding Active Directory.
3	423	43	104	644	4 exploit codes for vulnerabilities of Windows software found in 2020 including SMBGhost and CurveBall.
4	2306	117	478	7466	Exploit codes for vulnerabilities of OpenSSL including Heartbleed
5	1767	28	499	hidden	A exploit code for Log4Shell

mentioning a specific CVE ID and the number of exploit codes against the vulnerability, indicating that Twitter can be used to identify vulnerabilities that are likely to be exploited in the wild. Shrestha et al. have analyzed these three platforms in a similar way (Shrestha et al., 2020). They revealed that discussion on a specific software vulnerability can be found on GitHub even before it is officially published by NVD. Neil et al. have identified that GitHub can be used for mining threat intelligence regarding open source software (Neil et al., 2018). They represent obtained threat intelligence as a knowledge graph to notify security analysts and developers when any intelligence is found for software of their interests. One of the differences between our study and these existing studies is that we mainly focused on the aspect of GitHub as a platform for sharing exploit codes.

Analyzing Exploit Codes for Vulnerability Triage: Suciu et al. proposed the notion of expected exploitability to continuously estimate the likelihood that exploit codes will be published for a specific vulnerability (Suciu et al., 2022). They use features extracted from vulnerability information (e.g., NVD descriptions and CVSS scores), write-ups, and exploit codes collected from ExploitDB, Bugtraq, and Vulners to derive expected exploitability using neural network. Householder et al. have presented a historical analysis of availability of exploit codes for various vulnerabilities and investigated features of vulnerabilities that affect the likelihood that exploit codes are developed against them (Householder et al., 2020). Their study is similar to our study; however, they only use ExploitDB and Metasploit as sources of exploit codes and did not focus on GitHub. In (Jacobs et al., 2023), Jacobs et al. have described the engineering efforts for improving EPSS scores, which predict the probability each vulnerability will be exploited within next 30 days. They use the public availability of exploit codes targeting a specific vulnerability as one of the features for deriving its EPSS score. In 2013, Allodi et al. have revealed that ExploitDB covers 25% of vulnerabilities exploited in the wild (Allodi and Massacci, 2013), where their list of vulnerabilities expected in the wild only those referenced in the Symantec Attack Signature and Threat Explorer. Their results are similar to those we described in Section 3.3; however, they only focused on ExploitDB.

Detecting Malicious Codes on GitHub: Potentially malicious files such as malware and shellcodes can be uploaded to GitHub because it can host arbitrary text files, source code, and binary codes. Several existing studies have investigated these files. Rokon et al. have identified 7504 malware source code repositories on GitHub by using a machine learn-

ing model trained with manually labeled data (Rokon et al., 2020). The machine learning model uses repository metadata such as titles, descriptions, content of README files as features for classification. Cao et al. have analyzed malware repositories that have been published as forks of legitimate GitHub repositories to trick users into downloading the malware (Cao and Dolan-Gavitt, 2022). They used anti-malware software including ClamAntiVirus and CAPA to detect malware repositories and used the ssdeep algorithm to detect malware samples that are similar to already detected ones. Yadmani et al. have conducted a large-scale investigation of malware repositories that attempt to trick security researchers into executing them by making them appear to be repositories of exploit codes. Specifically, they have identified malware repositories using several heuristics, such as identifying codes that use IP addresses contained in malicious public IP address lists, analyzing binary executables with VirusTotal, and manual analysis of hexadecimal/base64-encoded payloads (Yadmani et al., 2023). They investigated exploit codes on GitHub like our study; however, they focused solely on GitHub and thus did not perform thorough comparison with other sources of exploit codes.

As represented by `CVE-2024-3094` (the vulnerability due to the backdoor intentionally implanted in XZ Utils), which attracted global attention in May 2024, developing efficient methods to prevent software supply chain attacks that add malicious code fragments to open-source software is an urgent task. A line of studies aims to detect malicious commits on GitHub. Gonzalez et al. proposed to use commit logs and repository metadata to identify malicious commits to a GitHub repository (Gonzalez et al., 2021). As a different approach, Gong et al. aim to detect malicious user accounts on GitHub by analyzing dynamic activities and interactions of users with deep neural networks (Gong et al., 2023).

Detecting Code Clones on GitHub: Code clone detection schemes have been used for various reasons including reduction of duplicated codes on a project, code provenance inference, and detection of the spread of bugs and vulnerabilities due to copy-and-paste of source codes. Some existing studies have investigated code clones on GitHub; however, they did not focus on exploit codes. Lopes et al. have conducted a large-scale investigation of code clones written in Java, C++, Python, and JavaScript by using SourcererCC (Lopes et al., 2017), and revealed that code clones are prevalent on the GitHub ecosystem. They have published the dataset of code clones; however, the dataset does not contain repositories published recently and thus insufficient for our investigation. Nguyen et al. have presented a framework for computing similarity of OSS projects on GitHub to recommend projects that facilitate on-going development (Nguyen et al., 2020). Zhang et al. have developed a recursive encoder-base machine learning model to detect cloned Java codes (Zhang and Wang, 2021). Wyss et al. have investigated code clones in npm packages and found 207 out of 6292 code clones carry vulnerabilities that originated from the original npm package (Wyss et al., 2022).

6 DISCUSSION AND CONCLUSION

In this study, we first compared exploit codes published on GitHub, ExploitDB, and Metasploit in terms of the numbers of exploit codes, coverage of vulnerabilities, coverage of exploits in the wild, and timeliness of exploit codes publication. As a result, we revealed that it is necessary to monitor exploit codes on GitHub together with the other sources for better vulnerability triage. We then proposed a scheme that combines existing code clone detection tools and a graph mining technique to extract exploit codes that worth in-depth analysis. We argue that this study is an important step toward improving the efficiency and reliability of vulnerability triage processes through automated exploit code analysis.

Finally, we discuss some limitations of this study. First, although we applied several heuristics to extract GitHub repositories with exploit codes in Section 2.1, repositories without exploit codes can be included in the final dataset. Thus, a promising research direction includes development of a machine learning model to determine whether a repository contains an exploit code or not for improving the validity. Second, we demonstrated how exploit code prioritization results can be used for vulnerability triage; however, vulnerability triage is commonly conducted by taking a wider range of information such as threat intelligence reports and vendor advisories into consideration. Therefore, another promising research direction is to combine the results obtained with our proposed scheme with other information to evaluate its usability for practical vulnerability triage.

REFERENCES

Allodi, L. and Massacci, F. (2012). A preliminary analysis of vulnerability scores for attacks in wild: the ekits and sym datasets. In *Proceedings of the 2012 ACM*

Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, BADGERS '12, page 17–24.

Allodi, L. and Massacci, F. (2013). Poster: Analysis of exploits in the wild. In *IEEE Symposium on Security & Privacy*.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007). Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591.

Cao, A. and Dolan-Gavitt, B. (2022). What the fork? finding and analyzing malware in github forks. In *Proc. of NDSS*, volume 22.

Chien, O. K., Hoong, P. K., and Ho, C. C. (2014). A comparative study of hits vs pagerank algorithms for twitter users analysis. In *2014 International Conference on Computational Science and Technology (ICCST)*, pages 1–6. IEEE.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Forum of Incident Response and Security Teams (FIRST) (2023). Common vulnerability scoring system version 4.0: Specification document. Accessed on Jun 10th, 2024.

Gong, Q., Liu, Y., Zhang, J., Chen, Y., Li, Q., Xiao, Y., Wang, X., and Hui, P. (2023). Detecting malicious accounts in online developer communities using deep learning. *IEEE Transactions on Knowledge and Data Engineering*, 35(10):10633–10649.

Gonzalez, D., Zimmermann, T., Godefroid, P., and Schäfer, M. (2021). Anomalicious: automated detection of anomalous and potentially malicious commits on github. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '21, page 258–267.

Gyöngyi, Z., Garcia-Molina, H., and Pedersen, J. (2004). Combating web spam with trustrank. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 576–587. VLDB Endowment.

Householder, A. D., Chrabaszcz, J., Novelly, T., Warren, D., and Spring, J. M. (2020). Historical analysis of exploit availability timelines. In *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. USENIX Association.

Hu, Y., Zhang, J., Bai, X., Yu, S., and Yang, Z. (2016). Influence analysis of github repositories. *SpringerPlus*, 5.

Jacobs, J., Romanosky, S., Suciu, O., Edwards, B., and Sarabi, A. (2023). Enhancing vulnerability prioritization: Data-driven exploit predictions with community-driven insights.

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97.

lazyhope (2023). Pythonclonedetection. Accessed on Jun 10th, 2024.

Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., and Vitek, J. (2017). Déjàvu: a map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA).

Neil, L., Mittal, S., and Joshi, A. (2018). Mining threat intelligence about open-source projects and libraries from code repository issues and bug reports. In *2018 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 7–12.

Nguyen, P., Rocco, J., Rubei, R., and Di Ruscio, D. (2020). An automated approach to assess the similarity of github repositories. *Software Quality Journal*, 28.

PoolC (2021). 1-fold-clone-detection-600k-5fold. Accessed on Jun 10th, 2024.

Rokon, M. O. F., Islam, R., Darki, A., Papalexakis, E. E., and Faloutsos, M. (2020). SourceFinder: Finding malware Source-Code from publicly available repositories in GitHub. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 149–163, San Sebastian.

Sabottke, C., Suciu, O., and Dumitras, T. (2015). Vulnerability disclosure in the age of social media: Exploiting twitter for predicting Real-World exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 1041–1056.

Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. (2016). Sourcerercc: Scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168.

sangHa0411 (2022). Python clone detection. Accessed on Jun 10th, 2024.

Schiappa, M., Chantry, G., and Garibay, I. (2019). Cyber security in a complex community: A social media analysis on common vulnerabilities and exposures. In *2019 Sixth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, pages 13–20.

Shrestha, P., Sathanur, A., Maharjan, S., Saldanha, E., Arendt, D., and Volkova, S. (2020). Multiple social platforms reveal actionable signals for software vulnerability awareness: A study of github, twitter and reddit. *PLOS ONE*, 15:e0230250.

Suciu, O., Nelson, C., Lyu, Z., Bao, T., and Dumitras, T. (2022). Expected exploitability: Predicting the development of functional vulnerability exploits. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 377–394, Boston, MA. USENIX Association.

Wyss, E., De Carli, L., and Davidson, D. (2022). What the fork? finding hidden code clones in npm. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2415–2426.

Yadmani, S. E., The, R., and Gadyatskaya, O. (2023). Beyond the surface: Investigating malicious cve proof of concept exploits on github.

Zhang, Y. and Wang, T. (2021). Cceyes: An effective tool for code clone detection on large-scale open source repositories. In *2021 IEEE International Conference on Information Communication and Software Engineering (ICICSE)*, pages 61–70.